Document Identification: common.proc.003
Document Revision: 1.1

Kabira Technologies, Inc.
IDLos Modeling Standards

DOCUMENT APPROVAL RECORD

Name	Title	Signature	Date
Dan Sifter	Chief Technical Officer		
Michael Lee	Vice President, Applications Engineering		
Joseph Fontaine	Director, North American Professional Services		

DOCUMENT SOURCE CONTROL

Document Type:	Adobe FrameMaker 6.0
Document File Name:	idlos_modeling_standards.book
Document Location:	engdoc/common/generic/techdoc

DOCUMENT CHANGE RECORD

Revision	Date	Description
Draft 1.0	September 02, 2003	Joseph S. Fontaine - First Draft
1.0	September 14, 2003	Joseph S. Fontaine - Incorporate comments from reviewers
1.1	October 30, 2003	Andy Fuchs - Incorporate additional discussion comments (Note: changes rekeyed 2005)

DISTRIBUTION LIST

Name	Title	Location
Dirk Epperson	Vice President, Product Strategy	San Rafael, CA
Michael Lee	Vice President, Applications Engineering	San Rafael, CA
Dan Sifter	Chief Technology Officer	San Rafael, CA
Peter Yu	Sr. Manager, Applications Development	Richmond, BC
Martyn Watkins	Director EMEA Professional Services	Berkshire, UK

REVIEWER LIST

Name	Title	Location
Michael Stroncek	Senior Consulting Engineer	Ballwin, MO
David Sundstrom	User Interface Development Manager	Austin, TX
Ken Cook	Applications Architect	San Rafael, CA
Michael Smus	Software Architect	British Columbia, CA
Andrea Canessa	Consulting Architect	San Rafael, CA
Amit Mahajan	Staff Applications Engineer	San Rafael, CA
Paul Johnson	Senior Technical Writer	San Rafael, CA
Otto Lind	Core Development Engineer	Lindstrom, MN
David Stone	Core Development Engineer	San Rafael, CA
Dan Sifter	Chief Technology Officer	San Rafael, CA
Bill Reinke	Applications Engineer	San Rafael, CA
Frederic Lionello	Applications Engineer	Paris, France
Loic Olichon	Technical Consultant	Paris, France

Table of Contents

1 Introduction

This document describes a collection of IDLos style conventions to be applied as standard practice in Kabira's engineering organizations. The goal is to provide uniformity across projects and organizations based on best practices; yielding models that are easy to understand, maintain, and enhance. A common set of standards will lead to greater consistency, productivity, and reduce the cost of maintaining legacy systems. As time passes experiences are gained and captured in this document.

An intentional distinction is made between "Modeling" and traditional "Coding" standards. A model is a pattern of something to be made helping to specify, visualize, and assess a particular problem space. A modeling language is the notation methods use to express designs and architectures that describe such patterns. IDLos is a textual modeling language, consistent with the Object Management Group's (OMG) Unified Modeling Language™ (UML™)[1]. Furthermore IDLos models are computationally complete[2]; thus maintaining fully executable properties while describing patterns in the semantics of the target subject matter (independent of concepts associated with classical programming such as distribution, transactions, threads, and memory management). Models are more abstract and often depicted graphically for ease of human understanding. As such, these style standards intersect, displace, and augment conventions applicable to classical programming techniques.

1. These standards shall be applied not only to production models but all IDLos models including unit tests, etc.

1.1 Scope

IDLos embodies the concepts of UML relevant to constructing executable models in distributed computing environments (roughly speaking this corresponds to constructs found in class diagrams, statechart diagrams, and action language specifications). A comprehensive modeling standard would also describe other software development lifecycle concepts (such as use-cases, sequence diagrams, collaborations, etc.). Additionally, Kabira frameworks such as KPSA and Streams generate the need for additional style standards. For the sake of tractability at this time, the scope of this document is restricted to IDLos-supported model constructs only, independent of any additional framework concerns.

In additional to textual IDLos specification ObjectSwitch applications may also be specified using UML graphical notation[3]; where appropriate this document addresses related conventions. Information described in the UML or IDL standards are not replicated within.

This document does not provide guidelines concerning organizational process models or development methodologies. In addition, it does not justify each convention (occasionally the motivation for a particular standard is presented, but this is bonus material).

1. Specifications to describe executable actions and procedures, including their run-time semantics, were recently added to UML in version 1.5.
2. To be precise, UML does not specify a single execution model; thus an IDLos model is computationally complete with respect to the additional semantics provided by the Kabira Infrastructure Server.
3. Graphical notations are automatically translated into IDLos via the Visual Design Center.

1.2 Conformance

This document provides both requirements and recommendations for IDLos-based model specification. Requirements are identified by the verb “shall”, recommendations by the verb “should”. Deviation of requirements are not permitted unless extremely irregular conditions exist; such cases shall be agreed upon by key project personnel and their justification documented. Deviation from recommendations are permitted but should be justified during review processes and documented in the associated model.

1.3 Audience

This document is primarily intended for Kabira’s Engineering organizations as a key component to a standardized process. Often, especially in professional services engagements, joint developments ensue with customers, system integrators, and partners; subsequently the writings are also applicable to these external organizations. Generally speaking, IDLos Modeling Standards apply to any ObjectSwitch-based development, and periodically serve as deliverables to organizations requiring evidence of software lifecycle data (DoD, FAA, etc.).

IDLos is based on OMG’s UML and IDL standards, discussions within assume a working knowledge of these technologies.

1.4 References

The following meeting(s), event(s) and/or material(s) influenced and/or contributed to the writing of this document.

1. “OMG IDL Style Guide”, Object Management Group, version ab/98-06-03, 1998.
2. “Executable UML”, Stephen J. Mellor, Marc J. Balcer, Addison-Wesley, 2002.
3. “Softwire Coding Standard”, Kabira Technologies, Inc., version 1.5, 2002.
4. “Code Conventions for the Java Programming Language”, Sun Microsystems, 1999.
5. “The Elements of Java Style”, Vermeulen, Amber, Bumgardner, Metz, Misfedldt, Shur, Thompson, Rogue Wave Software, Cambridge University Press, 2000.
6. “Netscape’s Software Coding Standards Guide For Java”, Christie Badeaux, Netscape Communications Corporation, 1999.
7. “Writing Robust Java Code”, The AmbySoft Inc. Coding Standards for Java, AmbySoft, Inc., version 17.01d, 2000.
8. “The Unified Modeling Language Users Guide”, Grady Booch, James Rumbaugh, Ivar Jacobson, Addison Wesley Longman, Inc., 1999.
9. “EMEA Design & Development Guide”, Antoine Jouannais, Kabira Technologies, Inc., version 1.7, November 14, 2001.
10. “Object-Oriented Systems Analysis: Modeling the World in States”, Sally Shlaer, Stephen J. Mellor, Yourdon Press, Englewood Cliffs, NJ, 1992.
11. “The Java Language Specification”, Gosling, J., Joy, B., Steele, G., Addison Wesley Longman Inc., 1996.
12. “ Java Coding Style Guidelines”, Sandvik, K., 1996.
13. “Kabira Process Improvement Initiative - Reuse Component”, Joseph Fontaine, July 14th, 2002.
14. “Object-Oriented Modeling and Design”, James Rumbaugh; Englewood Cliffs, NJ, Prentice Hall, 1991.

15. "Object-Oriented Requirements Analysis and Logistical Design: A Software Engineering Approach.", Donald G. Firesmith, Wiley, 1993.
16. "Object-Oriented Software Engineering: A Use Case Driven Approach", Ivar Jacobson, Magnus Christerson, Patrick Jonsson, Gunnar Overgaard; Addison-Wesley, 1992.
17. "Object-Oriented Software Construction", Bertrand Meyer, Prentice Hall, 1997.
18. "Tag Specification for Generating HTML-Formatted Documentation for IDLos Header", Peter Wong, Kabira Technologies, Inc., version 0.7, September 20, 2001.

1.5 Acronyms, Abbreviations and/or Terminology

The following acronyms, abbreviations are used this document.

Acronym/Abbreviation/Term	Definition
DBC	Design By Contract
UML	Unified Modeling Language
OMG	Object Management Group
IDL	Interface Definition Language

1.6 Typography Conventions

Typography conventions are as follows:

- Ordinary text appears in this font.
- `Filenames and IDLos constructs appear in this font.`
- **Rules are designated by unique numbers in this font.**

2 Files

2.1 File Names

2. All filenames shall be in lowercase.

3. Each package's behavioral model (action language constructs) should be captured in a single file with naming convention "`<packagename>.act`". If the package, for some reason is abnormally large where multiple files are warranted subsequent files shall be named according to logical content (e.g., `<packagename>_<content>.act`). These additional filenames shall contain underscores separating the package form the logical content name.

4. Each package's non-externalized structural model shall be captured in a single file with naming convention "`<packagename>.soc`".

5. If IDLos is being hand-generated, each package's externalized interface specification shall be captured in a single file with naming convention "`<packagename>.sdl`".

6. Each package's unit test code shall parallel the above only with the word test "test" prefixing each filename (e.g., `test<packagename>.sdl`, etc.).

7. If IDLos is being tool generated, each package's interface specification may be captured in either a "`.soc`" or "`.sdl`" file (tool specific choice).

8. Each package's component specification shall be captured in a single file with naming convention "`<componentname>.osc`".

9. Each package's deployment specification shall be captured in a single file with naming convention "`<componentname>.kds`".

2.2 File Structure

This section describes a standard file/directory structure for a project. Currently the "meta-build" initiative is in progress which will alter this structure in the future; the descriptions herein are to provide guidance for projects that begin in the interim.

10. Each "project" shall define a codeline. A project contains a set of components.

11. The project node shall be at the root of the codeline.

12. User trees shall split below the project node. In practice this de-marks the $SW_HOME environment variable for development environments using Kabira's codeline management tools.

```
./<project>/<user>
```

13. Component nodes shall exist directly below the project node as follows:

```
./<project>/<user>/<component1>
./<project>/<user>/<component2>
...
./<project>/<user>/<componentN>
```

14. A project that has system tests shall also host a "`systest`" directory adjacent to the component subdirectories. The `systest` directory shall host all end-to-end black-box tests.

```
./<project>/<user>/systest
```

15. A project shall host a "release" directory adjacent to the component subdirectories. The release directory shall contain built component archives and all other artifacts required for bundling into a release. This directory shall not contain any configuration controlled elements, it is transient and constructed when user trees are built.

```
./<project>/<user>/release
```

16. Each `component`, `systest`, and `release` directories shall have a “generic” platform directory, as well as a directory for each supported platform (“sol” for solaris, “hpux” for HP Unix). Files that apply across all platforms shall reside under “generic”, platform specific files shall reside under their respective directory.

```
./<project>/<user>/<component1>/generic
./<project>/<user>/<component1>/sol
./<project>/<user>/<component1>/hpux
```

17. Each platform directory shall host a “`src`” subdirectory for all modeled source files.

```
./<project>/<user>/<component1>/<platform>/src
```

18. The “test” subdirectory shall host a file named “`makefile`” that will build the test component (to be married with the host component built in “`src`” via deployment specifications as an executable unit test).

19. Each platform directory whose components have unit tests shall host a “test” subdirectory for all source files:

```
./<project>/<user>/<component1>/<platform>/test
```

20. The “test” subdirectory shall host a file named “`makefile`” that will build the unit test with a default target and run the unit test with a “`runtest`” target.

21. Each platform directory whose components have documentation (beyond that of `idlosdocs`” shall host a “docs” subdirectory for all source files:

```
./<project>/<user>/<component1>/<platform>/docs
```

22. Each platform directory may host additional subdirectories that are particular to the component (for custom native libraries, component-specific documents, etc.).

23. Each project shall host a “docs” directory for project-wide documentation (e.g., Users Guides, System Requirements, etc.).

```
./<project>/<user>/docs/<platform>
```

2.3 File Header

24. Each file shall contain a header with the following sections:

- File internal identification including filename, source code revision, date, and configuration source code location (the latter three automatically maintained by CVS).
- Copyright notice, use Kabira Technologies, Inc. or customer as appropriate.

25. Single line comments shall be used in the header with a comment delimiter appropriate for the file type.

An example IDLos file header is as follows:

```
// Name
//      <packagename>.sdl
//
```

```
// History
//      $Revision: $ $Date: $
//      $Source: $
//
// Copyright
//      Confidential Property of Kabira Technologies, Inc.
//      Copyright 2002, 2003 by Kabira Technologies, Inc.
//      All rights reserved.
//
```

An example `makefile` header is as follows:

```
# Name
#       makefile
#
# History
#       $Revision: $ $Date: $
#       $Source: $
#
# Copyright
#       Confidential Property of Kabira Technologies, Inc.
#       Copyright 2002, 2003 by Kabira Technologies, Inc.
#       All rights reserved.
#
```

3 Formatting

Formatting rules presented in this section apply universally and are referenced in subsequent sections when formation rules and statements are presented for each IDLos construct[1].

3.1 General

26. The first IDLos statement shall be flush left.

27. Each indention level shall be created by a tab character (never spaces).

28. Tab stops shall evaluate to 8 characters.

29. Line length shall be limited to 80 characters in all cases.

30. Code lines shall not wrap an 80 column display when tab stops are expanded to 8 characters.

3.2 White Space

31. If a single statement is to be continued across multiple lines, each subsequent line shall be indented one level from the initial line. The one exception to this rule is a rare case in rule 61.

32. A single blank line, and optional comment block, shall be used to separate logical groups of code within a method.

33. A single blank line shall be used to separate method definitions in behavioral specifications. No comment block is necessary.

34. Operators shall be bordered with a space on both sides (or if wrapping line, a newline on the right side is used in lieu of a space).

35. Parenthetically delimited expressions shall not have any whitespace between expression terms and the parentheses.

36. Operation/Macro invocations shall not have any whitespace between the enclosing parentheses and their arguments unless the resultant code is compiled out in production builds[2], in which case there shall be a single space between the enclosing parentheses and arguments.

37. Keywords shall always be followed by a single space unless a) it is at the end of the statement (e.g., `nolock`) or at the end of a line that is continued on following lines.

38. There shall be no trailing white space at the end of any line.

3.3 Simple Statements

39. Each line shall contain no more than one simple statement (a simple statement is an expression followed by a semicolon).

3.4 Compound Statements

Compound statements (also called blocks) are statements that contain lists of statements enclosed in braces "{statements}"[4].

1. For model peer reviews the modules are printed with "`cprint -l -s 6`".
2. Such cases include IDLos asserts and C++ macros.

40. The opening brace shall be on the next line after the line that begins the compound statement at the same indention level; the closing brace shall begin a new line and be indented to the beginning of the compound statement.

41. The enclosed statements shall be indented one more level than the compound statement.

Example:

```
{
        statement1;
        statement2;
        ...
}
```

42. Compound blocks shall be nested to three levels or fewer.

3.5 if-then-else Statements

43. The “if” keyword and conditional expression shall be placed on the same line with a single space separating the keyword and expression.

44. Braces and enclosed statement blocks shall follow the universal conventions from section “Compound Statements” on page 7.

45. Else clauses shall exist on their own line between “true” and “false” blocks.

Example:

```
if (expression)
{
        statement;
}
else if (expression)
{
        statement;
}
else
{
        statement;
}
```

3.6 create Statements

Create statements instantiate a type to shared memory. Values may be assigned to attributes in the create clause (which is atomic with the create).

46. The create keyword shall be followed by a single white space then the local variable name. If the “value” clause is present it shall follow the local variable with a space on each side, followed by a parenthesis. If a single value is being bound it shall reside on the same line directly after the parenthesis. If multiple values are being bound each value shall be on its own line. The line with the last value includes the closing parenthesis. Colons separating attribute names from values shall have a single space on each side.

Example:

```
create firstClassObject;

create firstClassObject values (attribute1 : someValue);
create firstClassObject values (
        attribute1 : someValue,
        attribute2 : someValue);
```

3.7 for Statements

47. Form 1: The "`for`" keyword and initialization, conditional, and update expressions shall be placed on the same line with a single space following the "for" keyword and the semi-colon ";" expression delimiter.

48. Form 2: The "`for`" keyword shall be followed by the variable, "in", and type all on the same line and separated by a single space.

49. Braces and enclosed statement blocks shall follow the universal conventions from section "Compound Statements" on page 7.

Example:

```
// Form 1
//
for (initialization; condition; update)
{
        statements;
}

// Form 2
//
for customer in Customers
{
        statements;
}
```

3.8 while Statements

50. The "while" keyword and conditional expression shall be placed on the same line with a single space separating the keyword and expression.

51. Braces and enclosed statement blocks shall follow the universal conventions from section "Compound Statements" on page 7.

Example:

```
while (expression)
{
        statement;
}
```

3.9 try-catch Statements

52. The "`try`" keyword shall appear on its own line.

53. Braces and enclosed statement blocks shall follow the universal conventions from section "Compound Statements" on page 7.

54. Any number of "`catch`" phrases consisting of the "`catch`" keyword and the exception expression shall follow each on its own line.

Example:

```
try
```

```
{
    statement;
}
catch (Deadlock e)
{
    statement;
}
catch (ObjectDestroyed e)
{
    statement;
}
```

3.10 declare Statements

Local variables are defined by the “declare” keyword. Reference “Local Variables” on page 35 for local variable initialization rules.

55. Local variable declarations shall be limited to one per line.

56. Local variable names shall follow the "Mixed case, beginning character lowercase" rule.

57. Local variables shall be declared at the top of each method or block defining a new context (see “Compound Statements” on page 7).

58. Local variables representing first class types automatically construct to empty, action models shall not explicitly default such variables to empty.

59. Local variable type information shall directly follow the “declare” keyword separated with a single whitespace on both sides.

60. Group declares shall not be used. For example...

```
// Do NOT use group declares!!!
//
declare
{
       long   i;
       long   j;
};
```

61. In the unusual event a qualified variable is too long to meet rule 29 the variable shall be split across lines at the outer scope module definition with the second line indented two levels.

Example:

```
declare long temperature;
declare swbuiltin::Date date;
declare ::Parlay::org::csapi::fw::fw_access::
               trust_and_security::IpAPILevelAuthentication auth;
```

3.11 select Statements

62. Empty checks shall be made before referencing any “selected” variable unless it is impossible the object would not exist (e.g., by design or “`on empty create`”). If assurance is by design and empty checks are not applied the object must be asserted for empty.

63. Select statements shall be composed on a single line if it fits within the maximum number of columns. If it cannot fit on one line the select statement shall be broken either before the “from” keyword or before the “where” keyword. Select-using statements shall follow the same rules but can also be broken before the “using” keyword or before the “on” keyword. If a lock specification exists it shall be on the same line as the “where” clause unless it exceeds maximum line length in which case it shall be on its own line.

Example:

```
// Check for empty after select in most all cases
//
select nextItem from self->Item[next];
if (!empty nextItem)
{
}

select nextItem from self->Item[next]
        where (nextItem.sequenceId == sequenceId);

select nextItem
        from Item using bySequenceId
        where (nextItem.sequenceId == sequenceId);

// No need to check for empty here, object is guaranteed to
// exist after select
//
select nextItem from Item using bySequenceId
        where (nextItem.sequenceId == sequenceId) writelock
        on empty create;
value = nextItem.value;

// As a matter of practice singletons are by design often
// constructed at genesis and selected later.
// Here its ok to make the design assumption the instance exists
// and assert rather than explicitly checking for empty.
//
select aCentralResource from singleton CentralResource;
assert( !empty aCentralResource );
```

3.12 Properties

IDLos employs properties to various declarations to apply certain semantic variations for base characteristics. Examples include access types on interfaces (`createaccess`, `deleteaccess`, `extentaccess`), `singleton`, `abstract`, etc. Properties are specified as a comma separated list between square brackets.

64. Each property shall be declared on its own line.

65. If a single property is specified it shall be delimited on the same line.

66. If multiple properties are specified the open bracket and each property shall be on its own line, properties shall be indented with a single tab.

67. It shall be permissible to restate properties even if they are redundant given the default.

68. Any property assignment operators shall be accompanied by a single space on both sides.

69. Properties must be specified before any IDLos documentation comments for the `idlosdoc` tool to work properly (reference “IDLos Document Tags” on page 29)[1].

Example:

1. May not be true with the VDC and coming documentation system.

```
[ virtual ]
::swcore::Format createFormatter();

[
       createaccess = revoked,
       deleteaccess = granted
]
/**
@interfaceIsoFormatter
@abstract              Format ISO wire messages to internal
                       message format.
@discussion            etc...
*/
interface IsoFormatter : ::swcore::Format
{
       ...
}
```

4 Formation Rules

This section describes the formation rules for the following IDLos concepts:

- Identifiers
- Components
- Packages
- Modules
- Interfaces
- Entities
- Constants
- Typedefs
- Attributes
- Keys
- Relationships
- Structures
- Unions
- Enumerations
- Operations
- Exceptions
- States
- Signals
- Transitions
- Triggers
- Expose Clauses

4.1 Identifiers

An identifier is a single name that uniquely distinguishes various IDLos elements such as packages, modules, interfaces, entities, states, instances, etc. Different classes of identifiers share common as well as their own unique formation rules.

70. Full English descriptions shall be used as identifier names.

71. Identifiers shall consist solely of alphanumeric characters, no underscores.

72. Abbreviations may only be used in the following cases:

- Acronyms
- Local variables representing exceptions may be declared as "e" (exception handling is very common, the use of the letter "e" for a generic exception is considered acceptable [11][12]).
- Loop counters within a method (e.g., "`i`", "`j`", "`k`").
- Component names
- Package names
- Module names
- Whenever the abbreviation is an extremely well-known in English or computing vocabularies and will not be questioned by anyone (note, this does not include any vertical domain specific terminology). For example, "identifier" is commonly understood as "ID", transmission control protocol as TCP, etc. Use this exception sparingly and with consideration!
- Enumerated literal names (see "Enumeration" on page 22).

73. Names shall not be shortened by removing vowels (special emphasis to the abbreviation rule).

74. Identifiers shall not encode any type information (such as in hungarian notation).

75. Fields that represent collections (arrays, sequences, etc.) shall be given plural names. In general, do not use "List", "Vector", etc. For example, "`customers`", "`orders`", etc.

76. One of the following rule sets shall be applied to identifier names depending on the particular lexical element (as described in following paragraphs):

- Rule 1: All lowercase.
- Rule 2: Mixed case, beginning character uppercase. All words begin with an uppercase letter with the remaining letters being lowercase (including acronyms, only the first character shall be uppercase).
- Rule 3: Mixed case, beginning character lowercase. All words begin with an uppercase letter with the remaining letters being lowercase (including acronyms, only the first character shall be uppercase).

77. The first letter of an acronym shall be uppercase and its remaining characters lowercase unless a) the acronym comprises the first word of an identifier classified for "Mixed case, beginning character lowercase" rules or b) the identifier is classified for "All lowercase" rules; in both cases the acronym shall be entirely lowercase. Names such as `sqlDatabase` for an attribute, or `SqlDatabase` for an entity, are easier to read than `sQLDatabase` and `SQLDatabase`[7].

78. Identifiers within the same scope shall be unique when case is not considered (don't use names that differ only in case).

79. Identifiers shall not use global scope qualifiers (e.g., use `swbuiltin`, not `::swbuiltin`).

4.2 Components

A component is any standard, reusable, previously implemented unit that raises the abstraction level to enhance programming constructs for developing applications. In IDLos this definition is further refined to a collection of related packages, a build specification, and documentation that cooperatively participate to fully define a reusable product compatible with other ObjectSwitch components and tools. A component's name must be unique from all other components in the application (preferably unique world-wide[1]).

80. Component names shall follow the "All lowercase" rule.

81. Since components essentially define the reuse granularity in ObjectSwitch its often desirable to reduce the number of packages per component; in fact often a component consists of a single package. Components should consist of the minimum number of packages required to define a complete, logical, reusable unit.

Examples:

```
// Protocol adapter transmission control protocol component
//
component patcp
{
};

// Protocol adapter network manager component
//
component panetman
{
};
```

4.3 Packages

A large domain can be divided into several packages, each of which is simply a smaller, more manageable, part of a domain[2]. Packages allow the specification of groups of logically related entities.

82. Package names shall follow the "All lowercase" rule.

83. Package names shall be singular.

84. Packages shall adhere to the following principles[5]:

1. OMG sponsors a submittal process for module reuse, perhaps one day we'll have a similar facility at Kabira for open sharing of components (e.g., reference [13]).

- The Common Reuse Principle - A package consists of entities that you reuse together. Place entities you usually use together in the same package.
- The Common Closure Principle - A package consists of entities all closed against the same kind of changes. Combine entities that are likely to change at the same time into the same package.
- The Acyclic Dependencies Principle - The dependency structure between packages must be a directed acyclic graph; there must be no cycles in the dependency structure. Two packages cannot directly or indirectly depend on each other.

85. Maximize abstraction to maximize stability. The stability exhibited by a package is directly proportional to its level of abstraction. The more abstract a package is, the more stable it tends to be[5].

86. A single package declaration should exist for its exposed (interface) declarations file (`.sdl`) and its internal declarations file (`.soc`)[1]. The exception to this rule is if declarations are necessary strictly to facilitate testing, these should be partitioned into their own package declaration sections.

87. Packages shall specify groups of logically related entities.

88. Declarations within Packages and Modules should have the following internal structure and sequence (some sequencing dependencies may force this schema to be slightly altered on occasion):

- Typedefs
- Constants
- Natives
- Enumerations
- Structures
- Unions
- Exceptions
- Interfaces/Entities
- Modules
- Relationships,
- Expose Clauses

89. The package names shall follow the “package” keyword separated by a single space and an open brace on the next line. Each declaration shall be indented one level from the open brace (with consideration for module scope). Braces closing the package/module shall be on their one line.

Example:

```
// Chargeback AT&T conference call
//
package cbmapattcc
{
        module  tmfrmap
        {
        };
};
```

4.4 Modules

Partitioning packages into modules is a useful technique for organization, clarity, and avoiding

1. IDLos permits packages to span multiple declarations.

type conflict with large namespaces. Each module represents a unique namespace representing the lowest-level cluster consisting of one or more classes and capturing the relationships between them[14].

90. Module names shall follow the "All lowercase" rule.

91. Module names shall be singular.

92. Modules shall be defined only once in a package's interface specification (`.sdl`)[1].

93. Modules shall be defined only once in a packages internal declaration specifications (`.soc`).

94. The module name shall follow the "module" keyword separated by a single space. The remaining formatting shall follow rules described in "Compound Statements" on page 7.

Examples:

```
// Switch message field identifiers
//
package swfid
{
        // ISO message field identifiers
        //
        module iso
        {
        };

        // Visa message field identifiers
        //
        module visa
        {
        };

        // American express message field identifers
        //
        module amex
        {
        };
};
```

4.5 Entities

An entity should represent an abstraction of a set of things with common characteristics and related behavior. An entity is a uniquely-identified type abstraction of a set of logically-related instances that share the same or similar characteristics[14].

Choose good entity names to improve the readability and understandability of the model. Strive for names that are clear, direct, and honest. Use common names where these are, or can be made, well-defined. When there is confusion in common names for the layperson, use industry standard names. Use strong, everyday words with extended meanings in preference to vague, unnecessarily technical, or esoteric terms. Append adjectives to common short names to make the short names more precise. Barring better alternatives use made-up names, even if they are lengthy, if they get at the essential character of the abstraction better than more customary, but less precise terms. Avoid certain "abused words" in naming your classes—words that have many meanings or great context dependence[2].

95. Entity names shall follow the "Mixed case, beginning character uppercase" rule.

1. IDLos permits modules to span multiple declarations.

96. Entity names should be short nouns or noun phrases drawn from the vocabulary of the system being modeled[8].

97. Entity names shall be consistent with the plurality of its type (e.g., collections shall have plural entity names; else singular).

98. Entities shall be have “`Impl`” suffixes to distinguish an entity from its possible interfaces; the rule applies even if an entity doesn’t have an interface.

99. Entity names shall be unique from other identifiers within the package (preferably within the component, and if possible within the system).

100. Entity names shall follow the “entity” keyword separated by a single space, the opening brace on its own line, the entity declarations indented one level, with a closing brace and semi-colon alone on the final line. Any parent entity name shall be on the same line as the the specialized name unless the line exceeds maximum line length per rule 29, in which case the parent type shall be on the next line indented one level from the specialized type name.

Example:

```
entity CarComputerMaintenanceSystemImpl : CarComputerSystemImpl
{
};

entity CarComputerMaintenanceSystemImpl :
        SomeVeryLongPackageName::SomeVeryLongEntityTypeName
{
};
```

101. Entities shall have the following declarative arrangement:

- Forward declares
- Typedefs
- Natives
- Enumerations
- Structures
- Unions
- Exceptions
- Attributes
- Operations/Triggers
- Signals
- States
- Transitions

Examples:

```
entity CustomerImpl
{
};

entity LifeCyleImpl
{
};

entity SaleImpl
{
};
```

4.6 Interfaces

An interface is a specification of the boundary of something in terms of the possible interactions or properties that are visible across that boundary. In ObjectSwitch Interfaces indicate which parts of an entity are exposed outside of a package. An entity may have zero or more interfaces. Interfaces are related to entities by “expose” statements (e.g., “realize” relationships in UML).

102. Interface names shall follow the "Mixed case, beginning character uppercase" rule.

103. Interfaces shall take on the same name as the entity it is exposing minus the “`Impl`” suffix and optionally contain a unique qualifier. For example, entity “`CarComputerImpl`” may be exposed by interface “`CarComputerMaintenanceSystem`”, or “`CarComputerCruiseControlData`”. Here the interface names are unique due to their protocol bindings, but this is not necessarily the only motivation for multiple interfaces.

104. Specialized interface declarations shall specify the parent interface on the same line with a single space one each side of the colon separating the types unless the line exceeds maximum line length per rule 29, in which case the parent type shall be on the next line indented one level from the specialized type name.

```
interface CarComputerMaintenanceSystem : CarComputerSystem
{
};

interface CarComputerMaintenanceSystem :
        SomeVeryLongPackageName::SomeVeryLongInterfaceTypeName
{
};
```

105. Often only a single interface is ever defined for an entity, in which case there shall not be an additional qualifier on the name.

106. Interfaces shall have the following declarative arrangement:

- Typedefs
- Natives
- Enumerations
- Structures
- Unions
- Exceptions
- Attributes
- Operations/Triggers

107. All declarations in interfaces shall have standard `idlosdoc` comment headers professionally documented (see “Comments” on page 28). This text is the eventual source of documentation available to package users and should well define usage and design contracts.

108. Interface names shall follow the “interface” keyword separated by a single space, the opening brace on its own line, the interface declarations indented one level, with a closing brace and semi-colon alone on the final line. Any parent interface name shall be on the same line as the specialized name unless the line exceeds maximum line length per rule 29, in which case the parent type shall be on the next line indented one level from the specialized type name.

Example:

```
interface Customer
{
};

interface CarComputerMaintenanceSystem : CarComputerSystem
{
};

interface CarComputerMaintenanceSystem :
       SomeVeryLongPackageName::SomeVeryLongInterfaceTypeName
{
};
```

4.7 Constants

A constant describes a literal whose value is immutable (cannot change at runtime).

109. Constant names shall follow the "Mixed case, beginning character uppercase" rule.

110. The constant's type shall follow the "const" keyword separated by a single space. Insert tabs between the type and the constant's name such that all names are left aligned, followed by a single space, and equal sign, the literal, and a semi-colon all on the same line.

Examples:

```
const long            PenniesPerDollar = 100;
const string          CompanyName = "Kabira";
const float           Pi = 3.14159265;
```

4.8 Typedefs

Type definitions provide an alias for a previously known type and also used to declare a sequence or array type.

111. Typedef names shall follow the "Mixed case, beginning character uppercase" rule.

112. Attributes representing collections or aggregates shall be declared from a single type name rather than direct used of the complex type (i.e., use a typedef to declare a simple type name before applying to attributes or local variable definitions).

113. The type being renamed shall directly follow the "typedef" keyword separated with a single space. Tabs shall be inserted between the original type and its newly derived name such that all type identifiers are left-aligned.

114. No spaces shall exist between type qualifiers and the "sequence" keyword.

Examples:

```
typedef sequence<Customer>           Customers;
typedef long                         FileDescriptor;
typedef string                       RecentFiles[4];
```

4.9 Attributes

An attribute is any named property used as a data abstraction to describe its enclosing object[14]. In IDLos attributes are defined in entities (and optionally exposed in Interfaces).

115. Attribute names shall follow the "Mixed case, beginning character lowercase" rule.

116. Attributes shall follow the following normalization rules:

- An attribute (including any inherited attribute) shall never be “not applicable” for any instance of an Entity.
- An attribute (including any inherited attribute) shall represent a property shared by all object instances of its host entity.
- All atomic elements of compound attributes shall be characteristic of its host entity (that is to say, attribute normalization rules apply for all elements of an aggregate attribute).
- Each attribute that does not compose a key must represent a characteristic of the instance named by the key and not a characteristic of some other non-key attribute[2]. For example the following would violate this rule:

```
entity SalesPersonImpl
{
        attribute string       name;
        key KeyName            {name};
        attribute string       m_officeAssignedTo;
        attribute string       m_officeAddress;<- NO
};
```

117. Attributes exposed through interfaces intended for access and not mutation shall be declared read-only.

118. Attributes not exposed in any interface must be prefixed with m_ to identify them as private members. Attributes exposed in an interface must not bear this prefix.

119. The type designator shall directly follow the “attribute” keyword separated with a single space. Tabs shall be inserted between the type and the attribute’s name such that all attribute identifiers are left-aligned.

Examples:

```
attribute string              name;
readonly attribute long       fileDescriptor;
```

4.10 Keys

A key is any set of one or more attributes, the values of which uniquely identify exactly one member of an extent[15]. Keys may be simple (consisting of a single attribute) or compound (consisting of multiple attributes). A key uniquely identifies an instance of an entity.

120. Key names shall follow the "Mixed case, beginning character uppercase" rule.

121. Key names shall be prefixed by the word “Key”.

122. One-part keys shall be suffixed by the name of the key’s attribute with its first character in uppercase (e.g., `KeyName`).

123. Multi-part keys shall be suffixed by the name of the key’s attributes, each attribute name converted to uppercase, each attribute separated by the word “And” (e.g., “`KeyDriversLicenseAndState`”).

124. The key name shall directly follow the “key” keyword separated with a single whitespace. Tabs shall be inserted between the type and the attribute’s name such that all key attribute compositions are left-aligned.

Example keys:

```
attribute string              name;
attribute string              socialSecurityNumber;
attribute string              address;
key     KeySocialSecurityNumbersocialSecurityNumber;
```

```
        key     KeyNameAndAddress      {name,address};
```

4.11 Relationships

A relationship is any potential logical connection between two or more things[15]. In IDLos relationships define associations between entities or interfaces. Each relationship (known as an “Association” in UML) has a name but is never referenced in IDLos apart from its declaration[1].

125. Relationship names shall follow the "Mixed case, beginning character uppercase" rule.

126. Relationship names shall provide a meaningful description of the semantics of the relationship.

127. Role names shall follow the "Mixed case, beginning character lowercase" rule.

128. Relationships should be uni-directional unless collaboration occurs in both directions.

129. The relationship name shall follow the “relationship” keyword separated by a single space, the open brace on its own line, each role specification on its own line, and the closing brace/semi-colon on its own line. Roles shall be indented one level with tabs. Each lexical element (excluding semi-colons) shall be separated by a single space.

Examples:

```
relationship CustomerImplToAccountImpl
{
        role CustomerImpl hasAccount 1..* AccountImpl;
        role AccountImpl assignedTo 1..1 CustomerImpl;
};
```

4.12 Structures

A structure is any heterogeneous aggregate type that groups an arbitrary number of other types.

130. Structure names shall follow the "Mixed case, beginning character uppercase" rule.

131. The struct name shall follow the “struct” keyword separated by a single space, the open brace on its own line, each structure member on its own line, and the closing brace/semi-colon on its own line. Structure members shall be indented one level with tabs. Structure members shall have tabs inserted between the type and the member name such that all names are left-aligned.

Example:

```
struct Result
{
        Messages                messages;
        boolean                 status;
};
```

4.13 Union

A union is a structure capable of holding different types at different times. Use unions only for

1. This is not necessarily the case for all action languages, and IDLos may be change in this regard.

compliancewith requirements of external interfaces.

132. In any case where a union name is not dictated by external requirements, the name shall follow the "Mixed case, beginning character uppercase" rule.

133. The union name shall follow the “union” keyword separated by a single space, followed by the keyword “switch”, followed by a single space, followed by an opening parenthesis, followed by the variant type name, followed by a closing parenthesis, the open brace on its own line, each union member on its own line, and the closing brace/semi-colon on its own line. Union members shall be indented one level with tabs. There shall be a single space between “case” selectors and the union’s alternatives, followed immediately by a semi-colon and a single space, the union member’s type, single space, then tabs to left-align member names.

134. Implementations using unions shall make no assumption about runtime’s storage allocation scheme.

Example:

```
union ResponseParameters switch (ResponseTypes)
{
	case negative: NegativeResponseParameters	negativeParameters;
	case positive: PositiveResponseParameters	positiveParameters;
};
```

4.14 Enumeration

An enumeration is a user-defined type whose instances are discrete named literals.

135. Enumeration names shall follow the "Mixed case, beginning character uppercase" rule.

136. Types that are inherently limited to a relatively small set of discrete possibilities should be typed as enumerations[1]. Deviations from this rule include cases such as when constants are employed to avoid “type mismatch” compiler warnings when aligning to a native API (by way of matching to intrinsic numeric type, the ObjectSwitch code generator does not map enumerated types to simple c++ enums).

137. Enumeration values shall follow the "Mixed case, beginning character uppercase" rule.

138. Enumerated values shall be unique to the package scope.

139. No assumptions shall be made about the code generator’s assignment of enumerated literal values other than the fact they will be unique.

140. Each enumerated value shall be declared on a single line of code.

141. The enumeration name shall follow the “`enum`” keyword separated by a single space. The remaining formatting shall follow rules described in “Compound Statements” on page 7.

Example:

```
enum TrafficLightColors
{
	Green,
	Yellow,
	Red
};
```

1. Strictly speaking, booleans, characters, fixed point numerics, and even floating point numbers in a computing system are formally large enumerated types but are not thought of as such in this context.

4.15 Operations

An operation is a service that may be requested from an object[16].

142. Operation names shall follow the "Mixed case, beginning character lowercase" rule.

143. Operation names should, in most cases, start with a strong, active verb. Operation names should be a short verb or verb phrase that represents some behavior of its enclosing type.

144. Operations that have no side-effects on the enclosing Entity's attributes shall be qualified with the "const" property.

145. Operations names that evaluate solely to a boolean return value shall have a meaningful verb prefix that implies a binary condition (e.g., `isLandingGearDown()`, `areFlapsUp()`, `canHandleRequests()`, etc.).

146. Operation names shall be meaningfully related to the enclosing entity name. A rule-of-thumb is a reader should not have to read anything more than the operation name to understand its intended function.

147. The development team may wish, within one or more packages, to discriminate one-way (asynchronous) from two-way (synchronous) operations by using a prefix for clarity. In this case, the one-way operations names should begin with the prefix "async", with the next character of the operation name in uppercase. When this practice is followed, it must be done uniformly throughout a package or not at all.

148. Action language shall not be placed in declarations. For example, don't do this...

```
entity Foo
{
        action bar
        {`
                ...
        `};
}
```

149. Generally speaking operations should contain at most 100 lines of code (comments excluded). This can't be an all encompassing rule, and the definition of LOC is somewhat subjective; but if a method exceeds this boundary it should come under scrutiny by the developer and review team. In most cases, is probably indicative of needing re-design.

150. Triggers shall have at most one operation associated with each trigger. Operations mapping to trigger specifications shall use the following conventions (in the following `roleName` and `attributeName` are changed such that the first character is uppercase):

- `create trigger - onCreate()`
- `delete trigger - onDelete()`
- `commit trigger - onCommit()`
- `abort trigger - onAbort()`
- `refresh trigger - onRefresh()`
- `relate trigger - onRelate<roleName>()`
- `unrelate trigger - onUnrelate<roleName>()`
- `pre-set trigger - onPreset<attributeName>()`
- `pre-get trigger - onPreget<attributeName>()`
- `post-set trigger - onPostset<attributeName>()`
- `post-get trigger - onPostget<attributeName>()`

Example:

```
// This example shows an unrelate trigger for role hasAccount
```

```
//
void    onUnrelateHasAccount();
trigger onUnrelateHasAccount upon unrelate hasAccount;

relationship CustomerImplToAccountImpl
{
        role CustomerImpl hasAccount 1..* AccountImpl;
        role AccountImpl assignedTo 1..1 CustomerImpl;
};
```

151. Lifecycle events shall have at most one operation associated to each event. Local entities hosting lifecycle events shall have a single entry point for each necessitated lifecycle event. Operations mapping to lifecycle properties shall use the following conventions:

- `initialize - onInitialize()`
- `recovery - onRecovery()`
- `terminate - onTerminate()`
- `load - onLoad()`
- `reload - onReload()`
- `packageinitialize - onPackageInitialize()`

152. The operation name shall follow the return type separated by a single whitespace, immediately followed by an open parenthesis. If there are no parameters close the parenthesis and add a semi-colon on the same line. Any parameters shall go on their own lines with a single space inserted between the parameter's type and name. The last parameters shall be followed with a parenthesis closing the parameter list. Any "raises" clause shall be on its own line with a single space between the "raises" keyword and exception clause.

Example:

```
void parse(
        inout swcore::Message swmsg)
        raises (ParseException);

void replace(
        in swcs::Version inactivating,
        in swcs::Versionactivating);

[ const ]
boolean isAlive();
```

4.16 Exceptions

An exception is any abstraction of an exception or error condition that is identified and raised by a method[15]. Exceptions represent conditions indicative of abnormal system behavior. Reference "Exceptions" on page 36 for discussions on practices on using exceptions.

153. Exception names shall follow the "Mixed case, beginning character uppercase" rule.

154. The exception name shall follow the "exception" keyword separated by a single space. If there are no members add one space, braces, and semi-colon on the same line. If members exist add a newline, an open brace on its own line, each member on its own line (with tabs inserted between the member's type and name such that all names are left aligned), followed with a closing brace and semi-colon it its own line.

Examples:

```
exception ParseError
```

```
        {
                string  errorCode;
                string  reason;
        };
```

4.17 States

States represent the various modes or life-cycle phase of an object during which certain rules of overall behavior apply. States represent a condition based on the cumulative history of an object. In IDLos the object enters a specified initial state upon genesis for which no behavior or automatic transition is permitted. In IDLos state machines are flat.

155. State names shall follow the "Mixed case, beginning character uppercase" rule.

156. The “`stateset`” keyword shall be on its own line, followed by an open brace on its own line, followed by each state (and comma) on their own lines, followed by closing brace and semi-colon on a single line.

157. Each state machine must have a single initial state named “Initial”. Transitions out of initial state must be explicit by way of a qualifying event. This initial state shall not (cannot) be re-entered.

Example:

```
        // Two-phase commit states for “coordinator” participant
        //
        stateset
        {
                Initial,
                Wait,
                Accept,
                Commit
        } = Initial;
```

4.18 Signals

Generally speaking a signal is any synchronous or asynchronous stimulation that is sent between two processes[16]. In IDLos signals refer specifically to operations on state machines hosted by active objects and are always asynchronous. Signals have no return or out parameters (in parameters are permitted). Signals sent to the same state must have the same signature. Signals cannot raise exceptions.

158. Signal names shall follow the "Mixed case, beginning character lowercase" rule.

159. The development team may wish, within one or more packages, to discriminate signals from operations by using a prefix for clarity. In this case, the prefix “sig” shall be used, with the next character of the signal name in uppercase. When this practice is followed, it must be done uniformly throughout a package or not at all.

160. The signal name shall follow the keyword “signal” separated by a single whitespace, immediately followed by an open parenthesis. If there are no parameters close the parenthesis and add a semi-colon on the same line. Any parameters shall go on their own lines with tabs inserted between the parameter’s type and name such that all names are left aligned. The last parameters shall be followed with a parenthesis closing the parameter list.

Examples:

```
        // Two-phase commit signals
```

```
//
signal commit();
signal voteCommit();
signal voteAbort();
signal done();

// Two-phase commit signals, prefix style
//
signal sigCommit();
signal sigVoteCommit();
signal sigVoteAbort();
signal sidDone();
```

4.19 Transitions

A transition is a rule that specifies what new state is achieved when an instance in a given state receives a particular signal[10]. When a transition fires an active object's state machine transitions to a new state. There are no automatic transitions or transition guards in IDLos.

161. Specify transition definitions on a single line; if the text is abnormally long refer to rule 31. Separate each lexical element on the transition specification by a single space.

162. Order transitions by source state, in the order the states appear in the stateset. Show each transition out of a given state prior to specifying transitions for the next state. Within each state, order the transitions by signal, in the order the signals are defined. Separate each source state's transitions from the next one's by a blank line.

Examples:

```
// This may not be exactly as desired to fulfill two-phase commit
// coordinator protocol but serve the illustration's purpose.
//
transition Initial to Wait upon commit;
transition Initial to ignore upon done;
transition Initial to cannothappen upon voteAbort;
transition Initial to cannothappen upon voteCommit;

transition Wait to cannothappen upon commit;
transition Wait to cannothappen upon done;
transition Wait to Abort upon voteAbort;
transition Wait to Commit upon voteCommit;

transition Commit to Initial upon done;
transition Commit to cannothappen upon commit;
transition Commit to cannothappen upon voteAbort;
transition Commit to cannothappen upon voteCommit;

transition Abort to cannothappen upon commit;
transition Abort to Initial upon done;
transition Abort to cannothappen upon voteAbort;
transition Abort to cannothappen upon voteCommit;
```

4.20 Triggers

A trigger is any method that executes when certain conditions occur. Triggers exist for certain entity, attribute, and relationship events.

163. Triggers shall be defined directly below the operation specified in the trigger.

164. Trigger definitions should be specified on a single line, if the text is abnormally long refer to rule 31. Each lexical element on the trigger specification shall be separated by a single space.

Examples:

```
void onCreate();
trigger onCreate upon create;

void onDelete();
trigger onDelete upon delete;

void calculatePayment();
trigger calculatePayment upon pre-get mortgagePayment;

void deleteCustomer();
trigger deleteCustomer upon unrelate patronizes;
```

4.21 Expose Clauses

Expose clauses specify which entity a particular interface exposes.

165. Expose clauses shall be specified on one line if the maximum line length rule 29 is not violated. Else two lines are used, the first line broken after the “with” keyword and the next line indented one level. Lexical elements on the same line are separated by single spaces.

Example:

```
expose entity IsoFormatterImpl with interface IsoFormatter;

expose entity SomeSuperLongTimeNameEntity with
       interface SomeSuperLongTimeNameInterface;
```

5 Documentation

5.1 Comments

Various comment forms are applied in IDLos specifications as described in the following sections.

166. Comments proceed the code they are characterizing (the comments do not follow the code).

5.1.1 Documentation Comments

Documentation comments take on the format `/** <comment> */` are processed by `idlosdoc` tooling (reference “IDLos Document Tags” on page 29). Refer to rule 69 for documentation comment positioning rules in relation to IDLos properties.

167. Interfaces shall use documentation style comments immediately before all declarations.

168. Documentation shall be described in third person narrative form. When describing purpose and behavior of entities, interfaces, and operations use pronouns such as “they” and “it” and third person verb forms such as “gets” and “sets” (instead of second person forms such as “set” and “get”) Third person verb forms that commonly appear are as follows[5]:

- adds
- allocates
- computes
- constructs
- converts
- deallocates
- destroys
- gets
- provides
- reads
- removes
- returns
- sets
- tests
- writes

5.1.2 Block Comments

169. Block comments of the form `/* <comment> */` shall not be used except for temporarily commenting out large blocks of code. Such cases shall be documented per rule .

5.1.3 Single Line Comments

170. Single line comments shall be used to document business logic, declarations apart from those in interfaces, and sections of code (i.e., all comments apart from documentation comments).

171. Single line comment indicator `"//"` shall be positioned as first two characters at the relative level of indention.

172. The first single line comment indicator in a block shall contain comment text. The last comment indicator shall stand alone on its own line.

173. End of line comments shall not be used (comments place on the same line as IDLos constructs).

Example:

```
// This is an example of a comment for separating code blocks.
// Always start with comment text on the first line of the block.
// The last line on a comment block shall be empty.
//
```

5.2 IDLos Document Tags

`IDLosdoc` parses special tags when they are embedded within a IDLos documentation-style comment block. These doc tags enable you to auto generate a complete, well-formatted API from your source code. The tags start with an “at” sign (@) and are case-sensitive - they must be typed with the uppercase and lowercase letters as shown. A tag must start at the beginning of a line (after any leading spaces and an optional asterisk) or it is treated as normal text. By convention, tags with the same name are grouped together[11]. Reference [18] for a comprehensive description of IDLos documentation tags. The following IDLos documentation tags are supported and required for documenting interface specifications.

TABLE 1. IDLos Document Tags

Tag	Used For	Purpose
`@abstract`	Packages Enumerations Interfaces Operations	Provides a very short synopsis of the construct being described.
`@attribute <name> <discussion>`	Attributes	Describes an attribute.
`@const <name> <discussion>`	Constants Enumerations	Describes the symbolic constant assigned to some immutable value.
`@copyright <discussion>`	Packages	Indicates that a package is copyrighted, the year that it was copyrighted in, and any descriptive information such as the name of the individual/organization that holds the copyright[7].
`@discussion`	Packages Enumerations Interfaces Operations	Provides a comprehensive description of the construct.
`@enum <name>`	Enumerations	Describes the name of an enumerated type.
`@example <discussion>`	Operations	Provides one or more examples of how to use a given operation. This helps developers to quickly understand how to use your entities[7].
`@exception <name> <discussion>`	Exceptions	Describes an exception type.
`@field <name> <discussion>`	Structures Unions Exceptions	Describes a field of an aggregate type.
`@interface <name>`	Interfaces	Describes the name of the interface.
`@operation <name>`	Operations	Describes the name of an operation.
`@package <name>`	Packages	Describes the name of the package.
`@param <name> <discussion>`	Operations Signals	Describes a parameter of an operation. Place the `param` tags in the same sequence as defined in the operation.

TABLE 1. IDLos Document Tags

Tag	Used For	Purpose
`@exception <exception> <discussion>`	Operations	Describes the exception and conditions under which it is raised within an operation.
`@relationship <name> <discussion>`	Relationships	Describes a relationship.
`@result <type> <discussion>`	Operations	Describes the information returned from an operation.
`@role <name> <discussion>`	Relationships	Describes a role of a relationship.
`@signal <name> <discussion>`	Signals	Describes a signal for an active object's state machine.
`@struct <name> <discussion>`	Structures	Describes a structure type.
`@typedef <name> <discussion>`	Typedefs	Describes a renamed type.
`@union <name> <discussion>`	Unions	Describes a union type.
`@warnings <discussion>`	Operations	Describes important usage rules, heuristics, and semantics associated with an operation.

6 Visual Modeling

This section describes the standards applicable when UML's graphical surface syntax is being applied to Kabira developments.

This section is TBD, I'll get to it later.

6.1 Realizes Relationships

174. Realizes relationships shall be drawn such that the arrow points from the entity to the interface consistent with the UML standard (i.e., the entity is realized by the interface)[1].

1. IDLos permits for the arrow to be drawn either way.

7 Practices

7.1 Assertions

Design by Contract (DBC)[17] is widely acknowledged to be a powerful technique for writing reliable software. The three key elements of DBC are preconditions, post-conditions and invariants. IDLos does not directly support DBC (an example language that does would be Eiffel). However assertions may be used as means for describing an operation's contract; particularly pre-conditions and post-conditions (invariants are more involved).

175. Code shall make extensive use of assertions to facilitate debugging, validating assumptions, checking preconditions, and catching unanticipated situations indicative of program errors[1].

176. Assertions shall not be used on data coming from the external world[2], external data must be observed for well-formedness by explicit checks and proper actions taken for abnormal data consistent with system failure mode requirements.

177. Assertions shall be used at the beginning of a method to check any pre-condition contract(s).

178. Assertions shall adhere to white space rules pertaining to development-only code per rule 36.

179. Assertions shall be used at the end of an method to check any post-condition contract.

180. Assertions shall not contain expressions that have side-effects or make function calls, e.g.,

- `assert( x+=1 );`
- `assert( updateMethod() );`

Example:

```
assert( !empty customer );
assert( status != Fail );
```

7.2 Dirty Associations

The author first heard this term coined in an action semantics committee meeting and really liked it (the discussion was whether or not dirty associations should be permitted in the action semantics meta-model, in the end they were much to the chagrin of the purists involved). Dirty associations refer to the practice of embedding an entity within an attribute as a navigation vehicle in lieu of a relationship. Such practices obscure the model, reduce readability, limit locking control/granularity, and cause UML practitioners to frown.

181. Dirty associations should not be used. In ObjectSwitch there is a small performance and memory footprint overhead for relationships that in some remote obscure case may justify a dirty relationship; unless such a scenario exists and agreed to by the project architect don't use them.

1. Note that assertions are compiled out in production mode.
2. Data that the program does not have absolute control over.

7.3 Unresolved Issues

Periodically IDLos implementations have unresolved issues that need to be described in a special set of comments.

The keyword "`FIX THIS`" shall be used to denote an unresolved issue. `"FIX THIS"` shall be followed by the initials of the person that documented the issue, the date the `"FIX THIS"` was scribed, the issue tracking identifier (if program is sufficiently mature where formal tracking policies are in effect) and a description of the issue as shown below:

```
// FIX THIS: JSF, September 01, 2003, ESP030901-000001
// Describe the unresolved issue here, the comment block
// shall be directly above the unresolved implementation.
//
```

182. If a program is sufficiently mature where formal tracking policies are in effect each `FIX THIS` shall be tracked in the issue tracking system. Multiple `FIX THIS` keywords may be added to the same issue.

7.4 Const Operations

IDLos provides the `const` property to indicate when an operation doesn't modify its object. Using this property tightens up the program and avoids problems when constant operations try to reuse operations that are semantically `const` but not declared as such (a constant operation cannot invoke a non-constant operation, so if its constant make it so).

183. All operation that do not have side-effects of the enclosing type shall be qualified with "`const`".

7.5 Accessors

Accessor operations are any small and simple method to get/set the value of an instance attribute.

184. Packaging/class design shall reduce the traditional need for accessors by minimizing (as much as possible) the data that needs to be externalized. Discretion shall be used when externalizing attributes, this should be kept to an absolute minimum (the perfect package has zero exposed attributes and operations;).

Most any reference will tell you to use accessors for operating on attributes in another class, even if its a higher order attribute residing in a superclass (the inference being "protected" attributes in classical object-oriented languages should never be used). In practice at Kabira we are not always so pure with IDLos, and for good reasons. First performance is always king and the code generator does not currently optimize out such operations at the action language level. Second, much of the classical accessor utility is minimized by a code generator that in fact constructs its own accessors "under-the-covers" (where all the runtime-related logic is added including resource mitigation, deadlock processing, etc.); these are things typical of accessors in traditional paradigms. ObjectSwitch has pre/post-get/set triggers available. We don't guarantee default values for lazy initialization. Finally if underlying maintenance was required the code generator would in general be able to take care of this for most cases, the model itself is only concerned about the abstraction not the implementation mechanics under the cover. Subsequently many of the stereotypical advantages of accessors in conventional paradigms are slightly less relevant in ObjectSwitch world. This prelude was submitted as not all the following rules would be considered conventional in mainstream practices:

185. Accessor functions (set/get) need not necessarily be used to set/get attributes where the attribute's entire instance and base data type will be externalized anyway.

186. Specialized types within a package may directly access attributes of higher-order types.

187. Accessor functions shall be used to get/set elements of externalized aggregate attributes; use a parameterized accessor to get/set the proper entry from the collection rather than exposing the collection.

188. When accessors are used they shall take on the name "`set/get<attributename>`" where the first character of the attribute name has been converted to uppercase.

7.6 State Machines

189. When using active objects construct a state transition table to assure proper intent is captured for all signal/state combinations in its state machine.

190. When exposing active objects outside of a package the state machine shall be designed such that "`cannothappen`" cannot happen (that is to say, assume the outside world is not adhering to any contract).

7.7 Promotional Deadlock Avoidance

191. Force an early write lock to avoid rollback if the design is such that contention can exist and a method, under most cases, will read lock then write lock a given object.

192. Use an explicit boolean attribute called "`writeLock`" to force write locks (the use of existing attributes obscure the intention and logic). Always set the dummy `writeLock` attribute to `true`.

Example:

```
action cbreport::ReportStatisticsImpl::incrementCdrCount
{`
        // Force write lock
        //
        self.writeLock = true;
        self.cdrCount += 1;
`};
```

7.8 Magic Numbers/Strings

A magic number is a bare naked number used in source code (the same concept applies for strings). It's magic because no-one has a clue what it means including the author outside of the first month. Bare naked numbers also imply any attempt to mask the violation in a symbolic manner (e.g., `const Ten = 10;`); make sure to use clear symbolic names (`e.g., const MaximumConnections = 10;`).

193. Magic numbers are prohibited from use and shall not be used, no exceptions.

194. "Naked" strings shall not be used, use a constant definition in lieu of embedding strings in action language.

7.9 Local Variables

195. Each local variable shall be used for one and only one purpose. Local variable names shall not be used for different purposes in the same method.

196. Local variable names shall be unique from other identifiers (such as attributes, operations, entities, etc.).

197. Local variables shall use explicit initialization with assignment operations in action language, rather than initializing on the declaration line.

7.10 Declaring Built-In Variables

Certain `swbuiltin` names are commonly used in ObjectSwitch applications.

198. The following common names shall be used:

```
declare swbuiltin::EngineServices engineServices;
declare swbuiltin::ObjectServices objectServices;
declare swbuiltin::TimeFunctions timeFunctions;
```

7.11 Return Values

Operations often use return values to indicate operation status.

199. Enumerations shall be used for operation status (as opposed to booleans, numeric values like -1, etc.).

200. If an operation returns a value, it shall always be checked by the caller.

201. Testing for operation “failed” status shall use the negative polarity of “success” in favor of a positive check for “failure” (to accommodate the possibility of additional literals).

Example:

```
// No - this needs to be maintained if more return
// failure cases are added.
//
assert( status == TimeoutFailure );

// Yes - no impact if more faild return codes are added
//
assert( status == Pass );
```

7.12 Print Statements

202. `”printf()”` shall never show up in any production code. Use event services, engine tracing, or some custom logging facility instead.

203. `”printf()”` statements may be used in unit test packages (these outputs become part of the permanent test log in “coordinator.stdout” and is consistent with the current test harness design).

204. “`printf(stderr,...)”` shall be used when `printf` is included in test code.

7.13 Precedence

205. Do not rely on implicit precedence rules in expressions. Expressions involving operators shall use explicit precedence specification via parentheses.

7.14 Exceptions

206. Exceptions shall only be employed solely for that purpose that is denoted by their name.

207. Exceptions shall be used strictly for “unexpected error conditions”.

208. Exceptions shall only be used in situations that are not typical for the normal program flow. Normal and expected cases shall be handled by function return values and/or transient parameters rather than exceptions.

209. Exceptions shall not be morphed into return codes where semantics are mixed (e.g., an operation may return “Fail” status indicating the operation failed for normal reasons, but the advent of an exception under abnormal conditions shall not be caught and translated to “Fail” as a return code. Here the exception must propagate.

210. Exceptions shall be thrown by an operation designed specifically for that purpose (as opposed to in-lining “throws” throughout the action language logic). This method shall be type void named “`throw<exception name>`” and shall have a single input argument for each exception member. Reasons for this are:

- It gives option to expose propagation (can optionally force propagation to runtime so callers can't catch it).
- Coerces people to breakdown exception and pass variances as arguments by a mandated signature.
- Centralizes redundant code and less in-line code in methods enhancing readability. e.g.,

```
self.throwParseError(
        lineNumber,
        errorCode,
        reason);

rather than this everywhere...

        declare ParseError e;
        e.lineNumber = lineNumber;
        e.errorCode = errorCode;
        e.reason = reason;
        throw e;
```

Example:

```
void throwParseError(
                in string errorCode,
                in string reason)
                raises (ParseError);
```

7.15 return Statements

Return statements are often abused, generally speaking a method should contain a single return statement. A typical deviation is when runtime integrity checks are made at the top of a method (to improve readability by minimizing extra nesting levels).

```
if (socket.state == Disconnected)
```

```
{
        return Failed;
}

...do stuff...

return status;
```

211. Apart from any initial integrity checks, methods should be constructed such that there is a single return statement (this is a very strong should).

212. Methods shall be structured such that there are no return statements beyond one level of indention.

7.16 Extentless Property

When operating on an extent (`create`, `delete`, `for`, etc.) "extentless" types do not lock the extent rendering improved performance over the alternative. This rule is obvious but placed here as more of a reminder for something that is often overlooked...

213. Extentless types should be used unless there is an imperative extents to be transactional.

7.17 Checking Enumeration Values

Enumerations define a finite set of literals, action language often uses conditionals to act upon each literal in the set; these constraints can become obsolete if new literals are added.

214. To check for complete coverage of enums in an `else if` structure, use an assertion in the final "`else`" clause :

```
// Do NOT use this construct!!
//
if (movie == Good)
{
}
else if (movie == Bad)
{
}
else if (movie == Ugly)
{
}
else
{
        swbuiltin::traceMsgFatal("Unexpected enum value " + movie);
        assert( false );
}

// Use this instead!!
//
if (movie == Good)
{
}
else if (movie == Bad)
{
}
else (movie == Ugly)
{
        assert (movie == Ugly);
        //do stuff for Ugly
}
```

8 Examples

8.1 Demonstration Package

The following illustrates a demonstration package that does nothing meaningful but illustrates every IDLos construct and document tags in proper form. Note some spacing may be slightly off as the word processor indents tabs differently than a text editor.

8.1.1 External Declarations Specification

```
// Name
//     idlosdemo.sdl
//
// History
//     $Revision: $ $Date: $
//     $Source: $
//
// Copyright
//     Confidential Property of Kabira Technologies, Inc.
//     Copyright 2002, 2003 by Kabira Technologies, Inc.
//     All rights reserved.
//

/**
       @packageidlosdemo
       @copyrightCopyright 2002, 2003 by Kabira Technologies, Inc.
       @abstractPackage abstract goes here, a couple lines is good.
       @discussionPackage description goes here.
                      Tab to left align each line.

                      This should be a fairly robust description.
                      This package does nothing meaningful, it simply
                      demonstrates well-formededness of IDLos constructs.
*/
package idlosdemo
{
       /**
       @typedefDemoTypedef
                             Describe the typedef here.
       */
       typedef sequence<long>DemoTypedef;

       /**
       @const         MaxDemoStructsDescription of constant goes here.
       */
       const longMaxDemoStructs = 10;

       /**
       @enum          EventIdentifiers
       @abstractEnumeration abstract goes here. <br>
              <h3><spacer type=horizontal size=20>
              HTML may be embedded into any of the documentation area
              <nobr>
       <a href="../../../somedirectory/somewebpage.html"target="_top">
              somewebpage.html
              </a></h3>
       @const         LiteralOne
                             The first literal.
       @const         LiteralTwo
                             The next literal, etc...
       */
```

```
enum DemoEnum
{
        LiteralOne,
        LiteralTwo
};

/**
@struct        DemoStruct
                       Describe the demo structure here.
@field         field1
                       Describe this field here.
@field         field2
                       Describe this field here.
*/
struct DemoStruct
{
        long   field1;
        string field2;
};

/**
@union         DemoUnion
                       Describe the demo union here
@field         variantAsStruct
                       Describe this field here.
@field         variantAsEnum
                       Describe this field here.
*/
union ResponseParameters switch (DemoEnum)
{
        case LiteralOne: DemoStructvariantAsStruct;
        case LiteralTwo: DemoEnumvariantAsEnum;
};

/**
@exceptionDemoException
                       Describe the demo exception here.
@field         errorCode
                       Describe this field here.
@field         reason
                       Describe this field here.
*/
exception DemoException
{
        long   errorCode;
        string reason;
};

[
        createaccess=revoked,
        deleteaccess=revoked,
        extentaccess=granted
]
/**
@interfaceDemoEntityInterface
@abstractInterface abstract goes here, a couple lines is good.
@discussionA robust description of this interface goes
               here.
*/
interface DemoEntityInterface
{
        /**
        @attribute demoAttribute1
                       Describe the attribute here.
        */
```

```
                attribute string demoAttribute1;

                /**
                @attribute demoAttribute2
                                Describe the attribute here.
                */
                attribute long demoAttribute2;
                key KeyDemoAttribute {demoAttribute2};

                /**
                @operationdemoException
                @abstractOperation abstract goes here.
                @param          errorCode
                                        Describe the parameter here.
                @param          reason
                                        Describe the parameter here.
                @result         void
                                        Describe operation return value here
                                        if not void.
                @exceptionDemoException
                                        Describe the conditions under
                                        which the exception is thrown.
                @discussionA crisp description of the operation goes here.
                */
                void throwDemoException(
                                in long errorCode,
                                in string reason)
                                raises (DemoException);

                /**
                @operationdemoSignal1
                @abstractOperation abstract goes here.

                @param          signalParam1
                                        Describe the parameter here.
                @param          signalParam2
                                        Describe the parameter here.
                @result         void
                                        Describe operation return value here
                                        if not void. Exposed signals are
                                        always void.
                @discussionSignals are always exposed via oneway voids.
                */
                oneway voiddemoSignal1(
                                        in string signalParam1,
                                        in long signalParam2);
        };


        /**
        @relationshipDemoRelationship
                                Describe the relationship here.
                                This example is a reflexive linked list.
        @role           predecessor
                                Describe the role here.
        @role           successor
                                Describe the role here.
        */
        relationship DemoRelationship
        {
                role DemoEntityInterface predecessor 1..1 DemoEntityInterface;
                role DemoEntityInterface successor 1..1 DemoEntityInterface;
        };
};
```

8.1.2 Internal Declarations Specification

```
// Name
//      idlosdemo.soc
//
// History
//      $Revision: $Date: $
//      $Source: $
//
// Copyright
//      Confidential Property of Kabira Technologies, Inc.
//      Copyright 2002, 2003 by Kabira Technologies, Inc.
//      All rights reserved.
//

/**
        @packageidlosdemo
                              This package is just for simply for
                              demonstration, it does nothing useful.
                              The current tools only run cleanly if
                              idlosdocs are defined in ".soc" file for
                              packages.
*/
package idlosdemo
{
        const stringDemoString = "demo";

        entity DemoEntityImpl
        {
                attribute stringdemoAttribute1;

                attribute long demoAttribute2;
                key KeyDemoAttribute {demoAttribute2};

                attribute string m_demoAttribute3;

                void throwDemoException(
                              in long errorCode,
                              in string reason)
                              raises (DemoException);

                void demoOperation();

                signal  demoSignal1(
                              in string signalParam1,
                              in long signalParam2);

                signal  demoSignal2();

                stateset
                {
                        Initial,
                        DemoState1,
                        DemoState2
                } = Initial;

                transition Initial to DemoState1 upon demoSignal1;
                transition Initial to cannothappen upon demoSignal2;
                transition DemoState1 to ignore upon demoSignal1;
                transition DemoState1 to DemoState2 upon demoSignal2;
                transition DemoState2 to DemoState1 upon demoSignal1;
                transition DemoState2 to ignore upon demoSignal2;
        };
        expose entity DemoEntityImpl with interface DemoEntityInterface;
};
```

8.1.3 Actions Specification

```
// Name
//      idlosdemo.act
//
// History
//      $Revision: $ $Date: $
//      $Source: $
//
// Copyright
//      Confidential Property of Kabira Technologies, Inc.
//      Copyright 2002, 2003 by Kabira Technologies, Inc.
//      All rights reserved.
//
action idlosdemo::DemoEntityImpl::throwDemoException
{`
        declare DemoExceptione;

        // Precondition - Error code must be > 0
        // (Place a space between parenthesis and expression terms
        // if the action is compiled out in production mode!).
        //
        assert( errorCode != 0 );

        // Bind diagnostic information to the exception
        // and throw it.
        //
        e.errorCode = errorCode;
        e.reason = reason;
        throw e;
`};

action idlosdemo::DemoEntityImpl::demoOperation
{`
        declare DemoEntityInterface demoInterface1;
        declare DemoEntityInterface demoInterface2;
        declare long numDemoEntities;
        declare long demoKey;
        declare long i;

        // Illustrates cardinalilty, a "for" loop, and
        // creating first class objects with multiple attributes.
        //
        numDemoEntities = cardinality(DemoEntityInterface);
        for (i=0; i<numDemoEntities; i++)
        {
                create demoInterface1 values (
                        demoAttribute1 : DemoString,
                        demoAttribute2 : i);
        }

        numDemoEntities = cardinality(DemoEntityInterface);
        i = 0;
        while (i < numDemoEntities)
        {
                // Keyed select with write lock and auto-create if instance
                // does not previously exist.
                //
                demoKey = numDemoEntities + i;
                select demoInterface1 from DemoEntityInterface
                        using KeyDemoAttribute
                        where (demoInterface1.demoAttribute2 == demoKey)
                        writelock
                        on empty create;

                // Select from an extent via relationship
                //
                select demoInterface2
                        from demoInterface1->DemoEntityInterface[successor]
                        where  (demoInterface2.demoAttribute2 == demoKey);

                if (empty demoInterface2)
                {
                        // Breaks out of iteration
                        //
                        break;
                }

                // Constraint would exceed maximum line length if on one line
                //
                if (demoInterface1.demoAttribute1 ==
                        demoInterface2.demoAttribute1)
                {
                        // Continues iteration
                        //
                        continue;
                }

                // Setup & tear down a bi-directional relationship (both
                // navigations are available once a relationship is established)
                //
                relate demoInterface1 predecessor demoInterface2;
                unrelate demoInterface1 predecessor demoInterface2;

                i += 1;
        }

        // There is many more examples that should be added here in the future
        // TBD.
        //
`};
```